OPTIMUM PARTITIONING OF A RECTILINEAR LAYOUT

LIANG-PING KU & HON WAI LEONG
Department of Information Systems and Computer Science
National University of Singapore, Lower Kent Ridge Road
Singapore 0511, Republic of Singapore
E-mail: kulp@iscs.nus.sg, leonghw@iscs.nus.sg

Abstract

Given a rectilinear layout consisting of n non-overlapping rectangles, we consider the problem of decomposing the remaining *free space* into rectangular *free blocks*. This problem is studied in connection with two problems from VLSI design, namely, design rule checking (DRC) using the corner stitching data structure, and the channel creation phase of global routing of VLSI building block layouts. In DRC using corner stitching, the decomposition is done via vertical partitioning. Similarly, horizontal partitioning can also be used. However, if we allow a hybrid of vertical and horizontal partitioning, the number of free blocks can be further reduced. In this paper, we consider the *optimum partitioning problem* in which we are interested in finding a decomposition that has the *minimum* number of free blocks. Given *any partition* of the free space, we first give a formula for counting the number of free blocks. The formula shows that the number of free blocks is determined by the number of *special* partition edges. We model the problem of finding an optimum partition as the problem of finding a maximum independent set in a bipartite graph. Finally, we give an $O(n^{2.5})$ algorithm for computing an optimum partition of any layout.

1 Introduction

We consider the partitioning of the free space in a rectilinear layout that consists of n non-overlapping rectangles, $R_1, R_2, \ldots, R_n$, placed in a rectangular region W. We call the region $\cup_{i=1}^{n} R_i$ the *covered region* and the region $F = W - \cup_{i=1}^{n} R_i$ (the complement) the *free space*. We are interested in the *partitioning problem* which is the problem of decomposing the free space F into (disjoint) rectangles called *free blocks*. This can trivially be done via either vertical partitioning (as illustrated in Figure 1) or horizontal partitioning (Figure 2). More generally, we can also consider a hybrid of vertical and horizontal partitioning (as illustrated in Figure 3).

The partitioning problem has a broad spectrum of applications. In VLSI layout, the problem arises in interactive tools for corner-stitching DRC [Ous84], plowing and compaction [Ous81]. It also arises in the channel creation phase of VLSI gridless routing [Que90] and global routing. The problem of vertical (or horizontal) partitioning has been applied to robot motion planning [LQ92], incremental 2-D bin-packing [Ku93, LK92], window placement problem [TAT91], and for 2-D memory management. A related problem of representation of the partition has also been studied in connection with VLSI floorplan design [Gra68, KK84, KK85, BS87, BS88, Kaj92].

Given a layout of the rectangles $R_1, R_2, \ldots, R_n$ in W, we first define the following notations: A corner of a rectangle R_j is said to be *active* if it does not touch any other rectangle. These active corners are concave corners of the free space F. Let $C = \{c_1, c_2, \ldots, c_a\}$ be the set of active corners. Then a general scheme for solving the partitioning problem is as follows: From *each* active corner $c_j \in C$, we extend a horizontal or vertical line (which we call a *partition edge*) until it hits another rectangle or the boundary or another partition edge. This will partition the free space into rectangular free blocks. Let $p(c_j)$ denotes the partition edge extended from the active corner c_j. Note that this will remove all concave corners from the free space F and so there is no need to consider the case in which we extend *both* a vertical *and* a horizontal partition edge from some active corners.

We are interested in the *optimum partitioning problem*, namely, the problem of partitioning the free space F into a *minimum* number of free blocks. In vertical partitioning studied in an earlier work [Ku93, LK92], only vertical partition edges are used and this results in a unique partition of

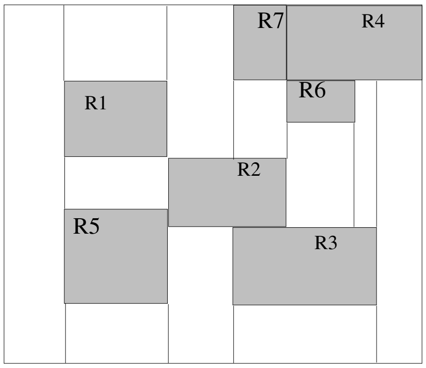


Figure 1: Vertical Partition

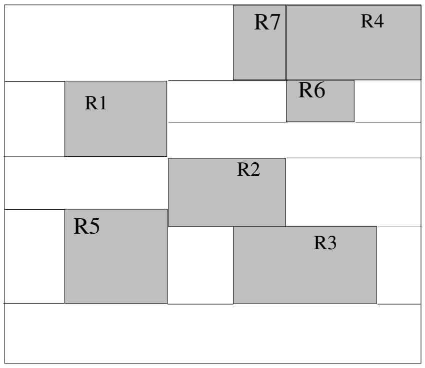


Figure 2: Horizontal Partition

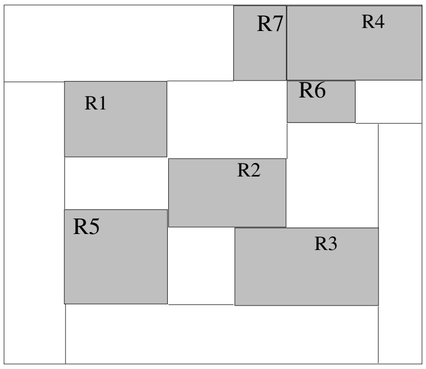


Figure 3: General partition

F. An example is shown in Figure 1 and the partition has 11 free blocks. Similarly, the same thing applies to *horizontal partitioning* as illustrated in Figure 2.

However, we can reduce the number of free block if we consider general partitions in which we have a hybrid of horizontal and vertical partition edges. An example is shown in Figure 3 which has 9 free blocks. In this case, it turns out that the number of free blocks depends on how we choose a "special" class of partition edges defined as follows: when $p(c_j) = p(c_k) = p$, namely, two partition edges are the same, we say that p is a *special partition edge*. An example is the partition edge between R_1 and R_5 in Figure 3.

In this paper, we present an $O(n^{2.5})$ algorithm for the optimum partitioning problem. We first derive a formula for counting the number of free blocks in any given partition of F. The formula shows that to minimize the number of free blocks, we need only to maximize the number of special partition edges. The latter problem of obtaining the maximum set of special partition edges can be transformed into the problem of finding a maximum independent vertex set in a bipartite graph.

The rest of the paper is organized as follows: In Section 2, we present the formula for counting free blocks in a given partition, and the formula is proven in Section 3. Section 4 shows the transformation of the optimum partitioning problem to the problem of finding a maximum independent vertex set while the full algorithm is presented in Section 5. In Section 6, a similar result [HLS93] is discussed and a flaw in their algorithm is presented. Finally, some concluding remarks are given in Section 7.

2 The Formula

For the simpler case of vertical partitioning (or horizontal partitioning), the formula to count the number of free blocks has already been studied [Ku93, LK92]. In this paper, we are interested in a general partition where partition edges can be in any direction. This is an important result as it provides the key to obtain the optimum partition.

We first present the intuition behind our formula. It has already been shown in [Ous84] that the number of free blocks in a vertical (horizontal) partition is bounded by $3n + 1$ (equal if all the sides of the rectangles has distinct x and y values). This result can easily be extended to the general partition. It can be observed that the number free blocks is proportional to the number of partition edges as each partition edge drawn would potentially decompose the free space. However, partition edges can be "saved" when $p(c_i) = p(c_j)$ (i.e. an occurrence of special partition edge) or when a corner c_k is inactive (i.e. solid rectangles touch each other). In the latter case, rectangles may touch each other in a complex manner, and hence we need to formalize the count of these touches.

Define a *line segment* to be a side of a rectangle R_i or the region W. Hence, the rectangles $R_1, R_2, \ldots, R_n$ and the region W is made up of four line segments each i.e. the left, right, top and bottom. Note that some of these line segments are collinear.

Let X_i be a vertical line that contains at least one vertical line segment, and Y_j be a horizontal line that contains at least one horizontal line segment, i.e. each X_i and Y_j holds a collection of collinear line segments. For example, Figure 4 shows all the X_is and Y_js of the layout. Let $n(X_i)$ ($n(Y_j)$ respectively) denote the number of line segments on line X_i (Y_j respectively) (see Figure 4 for illustration).

Define a *segment interval* to be the shortest interval on X_i (Y_j respectively) that contains all the line segments on X_i (Y_j respectively). Hence, each segment interval will have the form $I\gamma I\gamma I \ldots I\gamma I$, where I's are non-zero intervals which contain only line segments and γ's are non-zero intervals which do not contain any line segment. We call each γ a *normal gap*, and let $g(X_i)$ ($g(Y_j)$ respectively) denote the number of normal gaps on line X_i (Y_j respectively). For example, $g(X_4) = 1$ and $g(Y_2) = 1$.

Hence, the problem of counting number of touches between rectangles is reduced to counting number of overlaps between line segments within each I, which is the number of line segments in I minus one. Therefore, the general formula to count the number of horizontal touches is $H = \sum_j (n(Y_j) - g(Y_j) - 1)$.

However, the same formula cannot be used to count vertical touches. This is because some touches (for example in Figure 4, between R_1 and R_2, and R_6 and R_7) are limiting vertical touches as well as a limiting horizontal touches. Since the above formula has already counted them as horizontal touches, they should be excluded in the count for vertical touches. We, therefore, consider a vertical *imaginary gap* at these points. Defining formally, an *imaginary gap* is a zero interval (point) on a vertical line X_i, where ends of two vertical line segments meet such that these line segments belong to different sides of the rectangles they belong to. Let $g'(X_i)$ denotes the number of imaginary gaps on X_i. Hence the number of vertical touches is $V = \sum_i (n(X_i) - g(X_i) - g'(X_i) - 1)$.

Theorem 1 *The number of free blocks, f, in the partition is given by the formula:*

$$f = 3n + 1 - V - H - spe$$

where $V = \sum_i (n(X_i) - g(X_i) - g'(X_i) - 1)$
$H = \sum_j (n(Y_j) - g(Y_j) - 1)$
spe = *number of special partition edges in the partition.*

For example, in Figure 3, $V = 3$, $H = 6$ and $spe = 4$. Hence the number of free blocks $f = 3(7) + 1 - 3 - 6 - 4 = 9$.

3 Proof of the Formula

We now present the two major steps of the proof of Theorem 1. We first represent a partition as a planar graph G and then establish a relation between the number of free blocks and the number of edges in G. Next, we will count the number of edges in G in terms of V, H and spe. The proof follows from these two results.

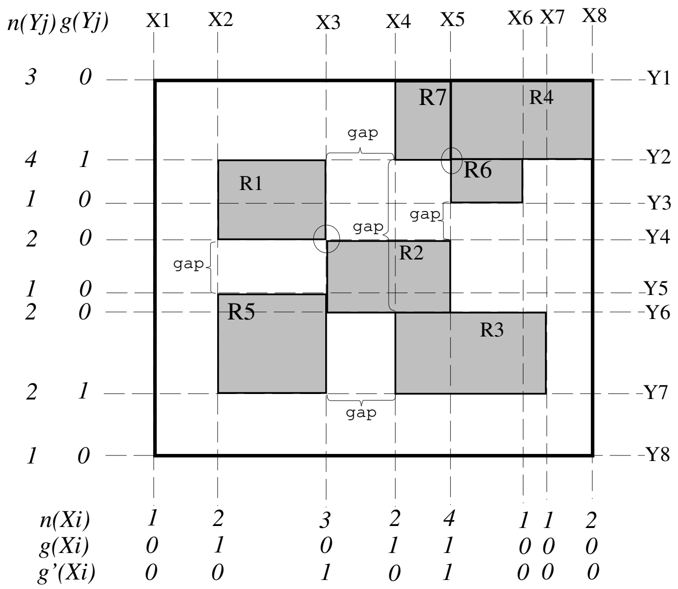


Figure 4: Examples of X_i and Y_j

3.1 Transforming Partition into a Planar Graph

Given any partition, it can be transformed into a planar graph $G(N, E)$ by doing the following transformation (Figure 5): Let all corners and intersection points be elements of N, and let all lines be elements of E. The dual graph $G'(N', E')$ of $G(N, E)$ is obtained by taking N' = internal faces of G, and $(n'_1, n'_2) \in E'$ if and only if n'_1 and n'_2 are adjacent faces in $G(N, E)$. Consider including imaginary edges into nodes in G that has degree of four (see Figure 5), we ensure that the dual graph G' is triangulated. We add four more nodes N_T, N_B, N_R and N_L, representing top, bottom, right and left side of the region W. There will be an edge from N_T, N_B, N_R or N_L to any node in G' if the corresponding face in the graph G touches the top, bottom, right or left boundary of the region W, respectively. In addition, five more edges (N_T, N_L), (N_T, N_R), (N_B, N_L), (N_B, N_R) and (N_T, N_B) are added to the graph yielding a graph $G^*(N^*, E^*)$ (see Figure 6). We have the following lemma that follows by construction.

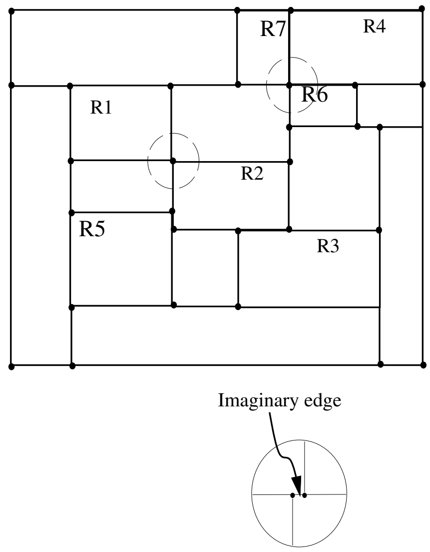


Figure 5: $G(N, E)$ representing the partition and insertion of imaginary edges

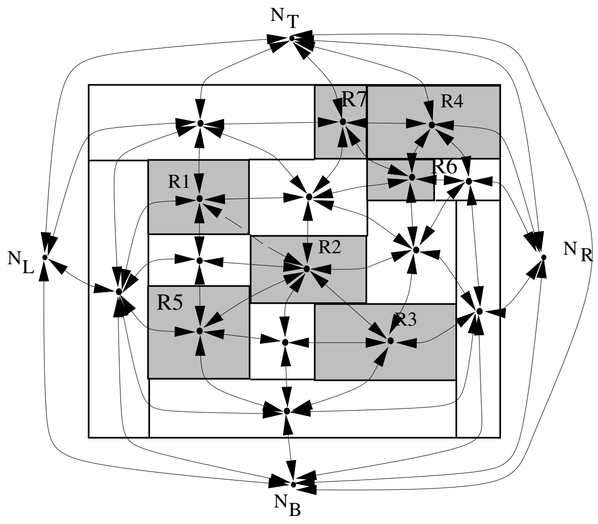


Figure 6: Maximal planar graph $G^*(N^*, E^*)$

Lemma 1 *The graph $G^*(N^*, E^*)$ is a maximal planar graph, i.e. all faces including the external face has degree of 3.*

With Lemma 1, we can now apply Euler's formula to give

$$|E^*| = 3|N^*| - 6 \tag{1}$$

All the nodes in G^*, with the exception of the four nodes (N_T, N_B, N_R, N_L) that are inserted, represent the faces in the graph G, which are also the faces of the rectangles and free blocks in the partition. Hence,

$$|N^*| = f + n + 4 \tag{2}$$

Further, each edge in G'', except the five that are added between the nodes N_T, N_B, N_L and N_R, correspond to an edge in G by the duality property. Hence,

$$|E^*| = e + 5 \tag{3}$$

where $e = |E|$ = number of edges in G (including the imaginary edges)

Combining Equation (1), (2), and (3), we get

$$\begin{aligned} e + 5 &= 3(f + n + 4) - 6 \\ f &= \frac{e-1}{3} - n \end{aligned} \tag{4}$$

For example in Figure 3, $e = 49$ (47 edges and 2 imaginary edges), hence $f = \frac{49-1}{3} - 7 = 9$.

3.2 Number of Edges e in Terms of n, V, H and spe

We now establish the relationship between e and n, V, H, and spe, namely,

$$e = 12n + 4 - 3V - 3H - 3spe \tag{5}$$

The general strategy is as follows: we first perturb the layout to obtain a simple case that can be proven trivially. We then collapse the perturbed layout in stages to get back the original layout. At each stage, we show the invariance of the Equation (5).

We will first perturb the rectangles $R_1, R_2, \ldots, R_n$ so that all the line segments (sides of rectangles) lie on distinct vertical and horizontal lines. This means that each X_i and Y_j will be split into multiple lines of X_{ik}'s and Y_{jk}'s, respectively. Each X_{ik} and Y_{jk} will contain exactly one line segment. Hence $V = H = 0$. Moreover, each partition edge $p(c_i)$ extended from the active corner c_i will be perturbed with respect to the new location of c_i. In the case of $p(c_i) = p(c_j)$ (i.e. a special partition edge), the equality do not hold any more, and hence it is split into two non-special partition edges $p(c_i)$ and $p(c_j)$ respectively. This results in $spe = 0$ after the perturbation. Furthermore, some corners c_k may become active after the perturbation and hence new partition edges $p(c_k)$ is introduced. For example, Figure 7 is the result of perturbation of Figure 3. All the four corners of R_4 are active, and the special partition edge between R_1 and R_5 is split into two non-special partition edges after the perturbation.

It can be seen that after the perturbation, the edge set, E, of the planar graph $G(N, E)$, is formed by the four sides of the region W, four sides of each solid rectangles R_i and the four partition edges from each of the four active corners of every solid rectangles. Each partition edge also cuts the edge that it hits into two as it will never hit a corner of another rectangle as a result of the perturbation. Hence, it can be shown inductively that $e = 12n + 4$.

We consider a collapsing step as moving two X_{ik}'s or Y_{jk}'s, dragging all the rectangles that has a side on it, until both lines coincide. When each collapsing step is executed, two cases which cause the count of e to decrease may occur:

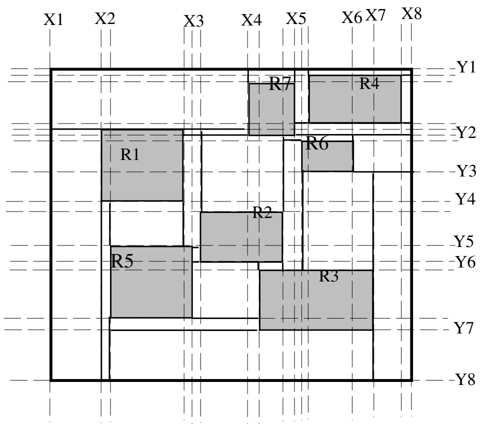


Figure 7: Partition after perturbation

- A corner c_k cease to extend a partition edge because it was an inactive corner before the perturbation, i.e. $p(c_k)$ vanishes as c_k becomes inactive in the collapsing step.
- Two partition edges $p(c_i)$ and $p(c_j)$ collapse into one special partition edge, i.e. $p(c_i) = p(c_j)$.

We can see that the first case will affect the count of V and H, and the second will affect the count of *spe*. We will prove the invariance of Equation (5) as we collapse the perturbed layout in stages back to its original layout in stages as follows:

Stage 1 Collapse only if V or H increases without introducing imaginary gap, followed by

Stage 2 Collapse only if no imaginary gap is introduced, followed by

Stage 3 Collapse if imaginary gap is introduced.

In the following sub-sections, we analyze each stage by listing out exhaustively all possible collapsing steps and show that the invariant holds (i.e. each increase in V, H, *spe* reduces e by three).

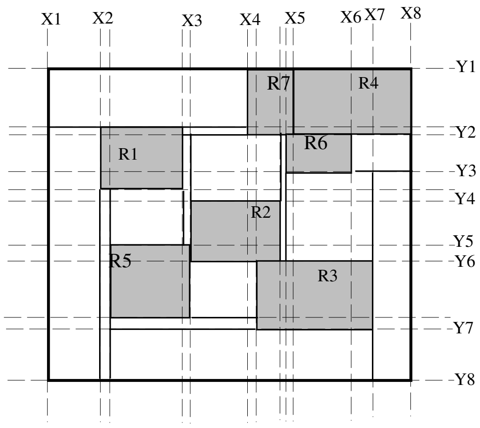


Figure 8: Partition after Stage 1

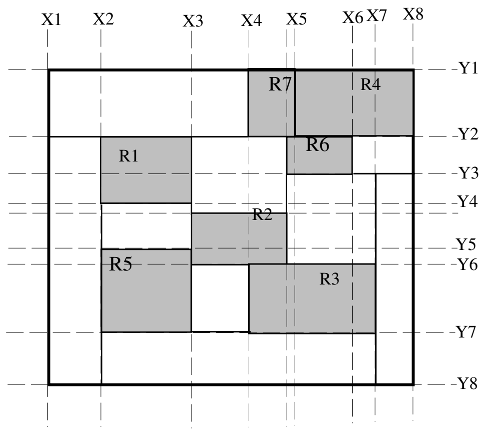


Figure 9: Partition after Stage 2

3.2.1 Stage 1 : Collapse that only increases V or H

We will now start to collapse the corresponding lines X_{ik}'s or Y_{jk}'s together as long it increases V or H, without introducing any imaginary gap. To reduce the number of possible cases, we first prove the following property that holds for each collapsing step:

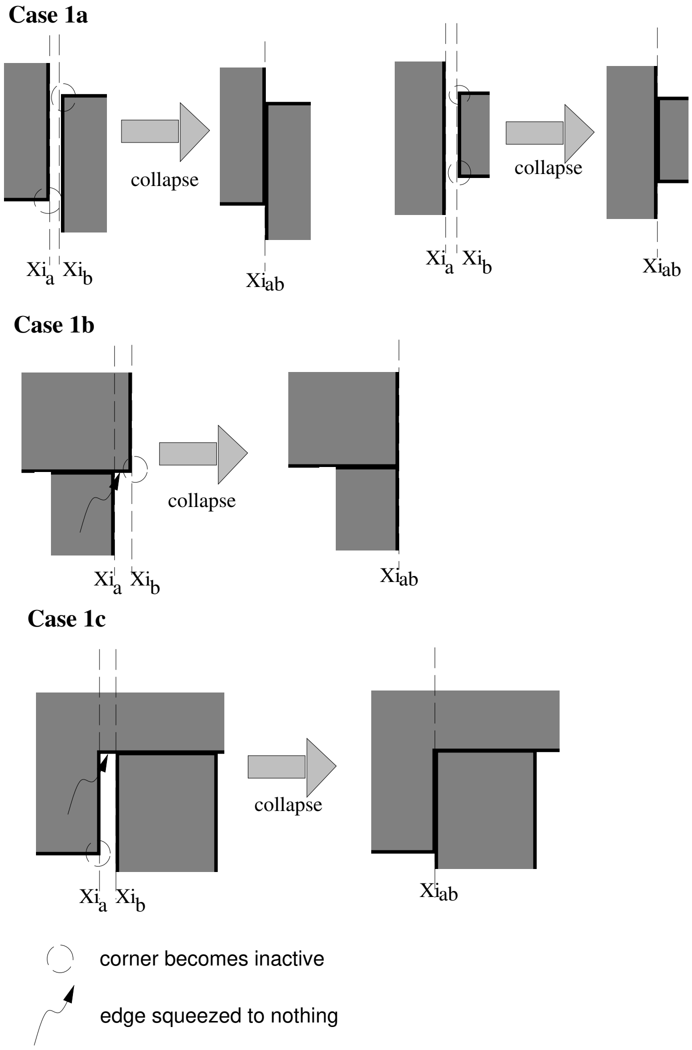


Figure 10: Cases where V increases by 1

Property 1 *In each collapsing step, no gaps (either normal or imaginary) will be formed and each collapsing step will only increase V or H by exactly one.*

We will prove by induction on the number of collapsing step as follows: Consider a particular X_i which has been split into $X_{i_1} \dots X_{i_s}$. So the basis is before any collapsing step, all X_{ik}'s has only one line segment, hence no gaps. Assume that at any moment of time, all X_{ik}'s has no gaps. Let V_i denotes the count of V that all X_{ik}'s contribute, i.e. $V_i = \sum_k (n(X_{i_k}) - 1)$. In the next collapsing step to increase V_i, let X_{i_a} and X_{i_b} be the chosen lines to be collapsed together into $X_{i_{ab}}$. The new count for V_i will be

$$\begin{aligned} V_i' &= \sum (n(X_{i_k}) - 1) - (n(X_{i_a}) - 1) - (n(X_{i_b}) - 1) \\ &\quad + n(X_{i_a}) + n(X_{i_b}) - 1 - g(X_{i_{ab}}) - g'(X_{i_{ab}}) \\ &= V_i + 1 - g(X_{i_{ab}}) - g'(X_{i_{ab}}) \end{aligned}$$

However, V must increase (i.e. $V_i < V_i'$), which implies that $g(X_{i_{ab}})$ and $g'(X_{i_{ab}})$ must be zero, and $V_i' = V_i + 1$. Similar proof can also be worked out on a collapsing step with two horizontal lines. Hence by induction, we have proved property 1. With property 1, we will list exhaustively all cases where V or H increases by exactly one when two X_{ik}'s or Y_{jk}'s are collapsed (see Figure 10 and Figure 11), and for each case, we will show the invariance of Equation (5).

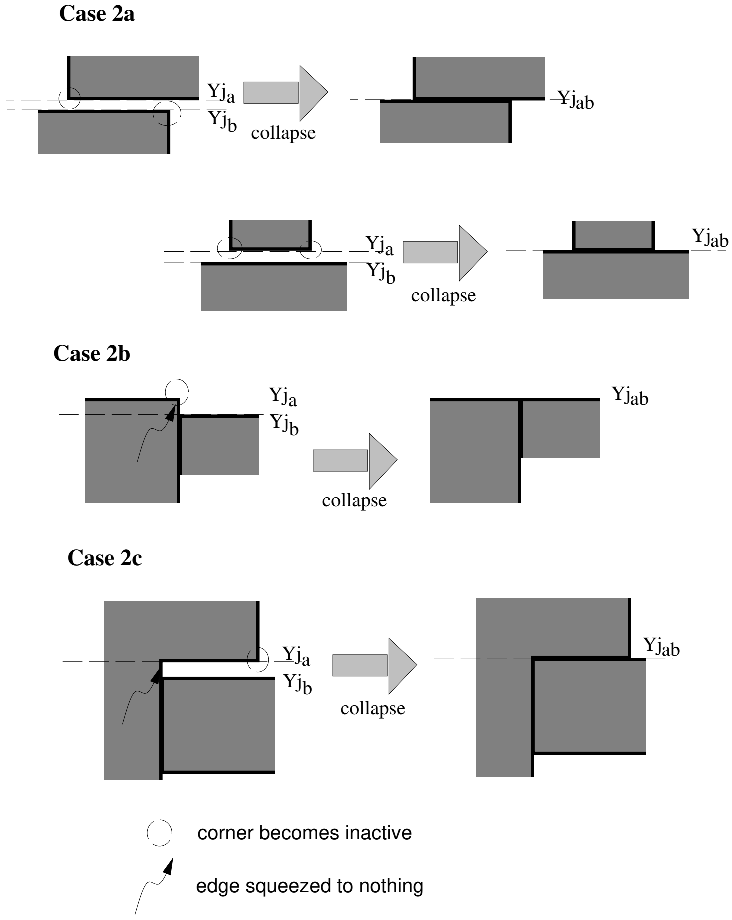


Figure 11: Cases where H increases by 1

Note that in contrary to perturbation, each collapsing step causes active corners to become inactive. Each active corner c_i indirectly contributes two in the count of e, i.e. the partition edge $p(c_i)$ and the edge that is cut into two by $p(c_i)$. Hence, when a active corner changes from active to inactive, value of e reduces by two.

Consider Figure 10, in case 1a, exactly two corners become inactive which causes e to reduce by four. However, an edge is introduced at the place where the two line segments (sides of the rectangles) overlap. Therefore, there is a net loss of three edges. In cases 1b and 1c, exactly one corner becomes inactive which causes e to decrease by two. Moreover in the collapsing step, a tiny horizontal edge is also squeezed away, hence making a total net loss of three edges. Therefore, each increase in V will reduce e by three.

All the three configurations could be rotated 90^o to form the cases in Figure 11. Since imaginary gap is not involved, the same reasoning can be applied to the three cases in Figure 11. Hence each increase in H will also reduce e by three.

3.2.2 Stage 2 : Collapse that does not introduce imaginary gaps

Consider the perturbed partition after underwent collapsing in stage 1. In this stage, we will collapse any pair of lines only if no imaginary gap is introduced. Again, we will prove the following property to reduce the possible number of cases.

Property 2 *In each collapsing step, exactly one normal gap will be introduced with no imaginary gap being introduced.*

We will prove by induction on the number of collapsing step as follows: Consider the vertical lines $X_{i_1} \ldots X_{i_s}$ that was split from X_i. So the basis is before executing any collapsing step in this stage, all X_{ik}'s has no gaps. Assume that property 2 holds at any moment of time. The count of V that all X_is contributes before the collapse is $V_i = \sum(n(X_{i_k}) - 1 - g(X_{i_k}))$. In the next collapsing step, let X_{i_a} and X_{i_b} be the chosen lines to be collapsed together into $X_{i_{ab}}$. The new count for V_i will be:

$$\begin{aligned} V_i' &= \sum(n(X_{i_k}) - 1 - g(X_{i_k})) - (n(X_{i_a}) - 1 - g(X_{i_a})) - (n(X_{i_b}) - 1 - g(X_{i_b})) \\ &\quad + n(X_{i_a}) + n(X_{i_b}) - 1 - g(X_{i_{ab}}) - g'(X_{i_{ab}}) \\ &= V_i + 1 - (g(X_{i_{ab}}) - g(X_{i_a}) - g(X_{i_b})) - g'(X_{i_{ab}}) \end{aligned}$$

Since there is no imaginary gap, and by the assumption that the partition has underwent stage 1 (i.e. $V_i' = V_i$), the number of normal gaps increase by one.

Similarly, considering a collapsing step in the horizontal direction will result in the new count of H_j as:

$$\begin{aligned} H_j' &= \sum(n(Y_{j_k}) - 1 - g(Y_{j_k})) - (n(Y_{j_a}) - 1 - g(Y_{j_a})) - (n(Y_{j_b}) - 1 - g(Y_{j_b})) \\ &\quad + n(Y_{j_a}) + n(Y_{j_b}) - 1 - g(Y_{j_{ab}}) \\ &= V_i + 1 - (g(Y_{j_{ab}}) - g(Y_{j_a}) - g(Y_{j_b})) \end{aligned}$$

This again shows that number of normal gaps increases by one and hence we have proved property 2.

However, complication may arise if a collapsing step destroys an existing gap and generate two new normal gaps (see Figure 12). This complication can be work around by ordering the collapsing steps as in Figure 13. Hence, we list exhaustively all simple cases where only one normal gap is introduced at each collapsing step (see Figure 14 and Figure 15).

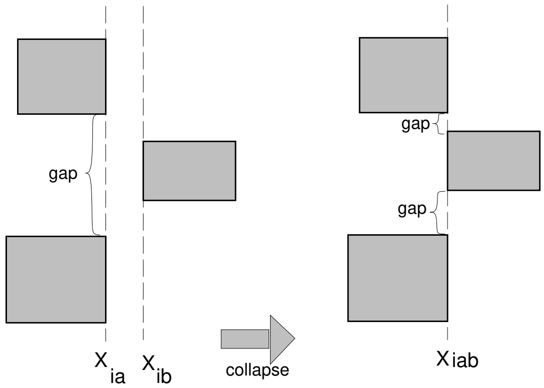


Figure 12: Existing gap broken up into two gaps

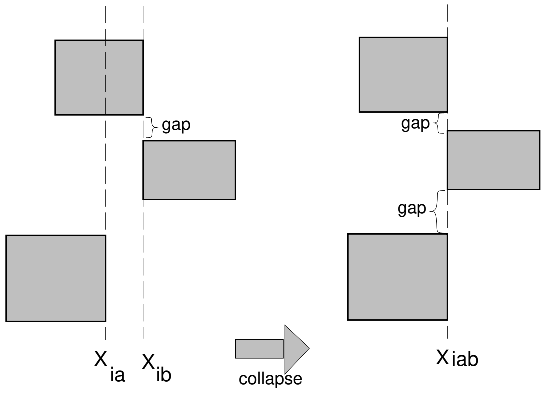


Figure 13: Ordering the collapsing steps

Observe that a special partition edge always lies along a normal gap, hence each collapsing step may increase *spe* by one. We will first analyze the change in e when a collapsing step introduces a special partition edge, and then when it does not.

In case 3a, the two partition edges $p(c_i)$ and $p(c_j)$ are merged into one to form a special partition edge. In this collapsing step, the four edges ($p(c_i)$, $p(c_j)$ and the two edges cut by them) vanishes and left with the merged edge of $p(c_i)$ and $p(c_j)$. Hence there is a net loss of three edges for each introduction of a vertical special partition edge.

Case 3b, 3c and 3d illustrate cases where no special partition edge is formed. In case 3b, it is obvious that there is no change in e. However in case 3c and case 3d, one edge is loss, but an imaginary edge is formed. Hence, there is no change in e. Note that the edge shown in the diagram is not a special partition edge since $p(c_i) \neq p(c_j)$.

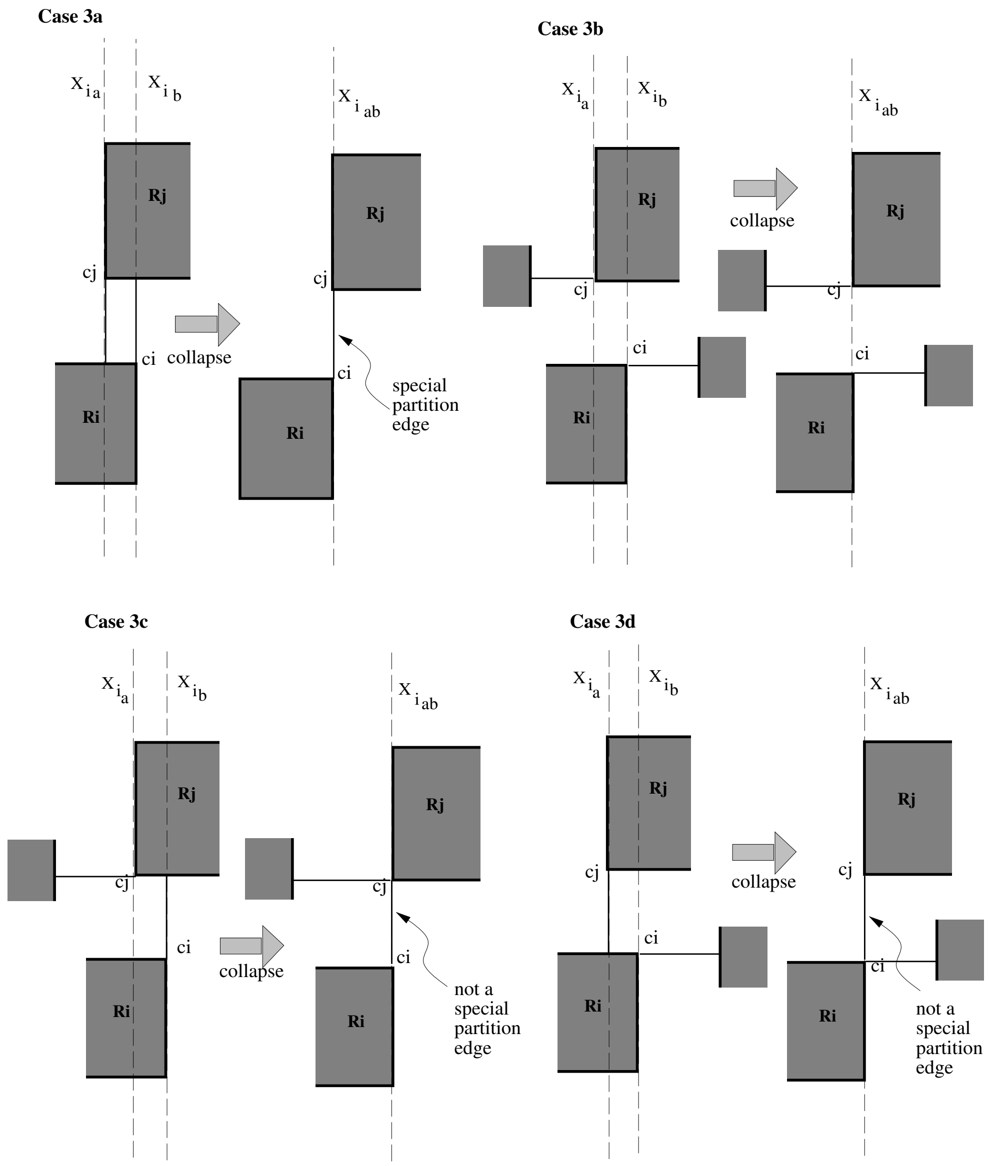


Figure 14: Collapse that introduces a normal gap

For each of the cases in Figure 14, we could derive the corresponding case which assumes collapsing from the other direction. These cases are labeled 3a', 3b', 3c' and 3d' respectively in Figure 15. The changes in V, H, e and spe are the same as those of their corresponding cases in Figure 14. All the above cases, if rotated 90^o, can be applied to the collapsing steps in the horizontal direction, and the analysis of the changes in V, H, e and spe will be identical to those in the vertical counterparts.

3.2.3 Stage 3 : Collapse that introduces imaginary gaps

Collapsing steps that introduce imaginary gaps are handled specially in this section because firstly, it is the only non-symmetrical component between the vertical and horizontal direction, and secondly, it may cause V to decrease or H to increase, Furthermore, it always introduces an imaginary edge.

Consider case 4a in Figure 16. When X_{i_a} and X_{i_b} are collapsed together, H will increase by one because the horizontal normal gap is squeezed to nothing. V will not change and this can be shown as follows using the same notation from the previous section:

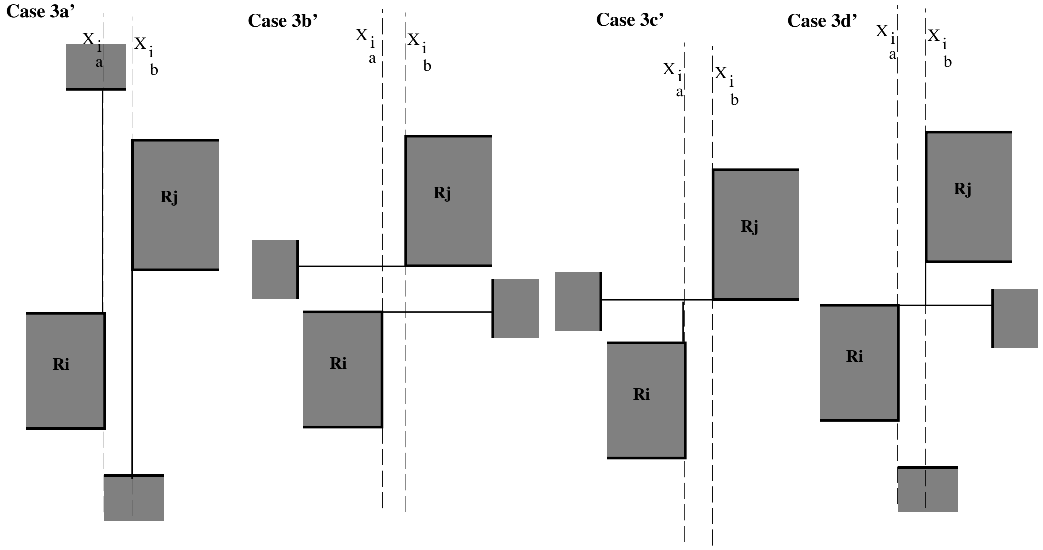


Figure 15: Collapse that introduces a normal gap; other cases

$$
\begin{aligned}
V_i' &= \sum(n(X_{i_k}) - 1 - g(X_{i_k}) - g'(X_{i_k})) \\
&\quad -(n(X_{i_a}) - 1 - g(X_{i_a}) - g'(X_{i_a})) - (n(X_{i_b}) - 1 - g(X_{i_b}) - g'(X_{i_b})) \\
&\quad +n(X_{i_a}) + n(X_{i_b}) - 1 - g(X_{i_{ab}}) - g'(X_{i_{ab}}) \\
&= V_i + 1 - (g(X_{i_{ab}}) - g(X_{i_a}) - g(X_{i_b})) - (g'(X_{i_{ab}}) - g'(X_{i_a}) - g'(X_{i_b})) \\
&= V_i + 1 - 0 - 1 \\
&= V_i
\end{aligned}
$$

In this collapsing step, two corners are made inactive and an imaginary edge is introduced. To analyze the change in e, we need to consider whether there is a special partition edge joining these two corners before the collapse. If there is, then there is no change in e since the removal of this special partition edge is balanced off by the insertion of a imaginary edge. So the increase in H merely compensates the decrease in spe. On the other hand, if there is no special partition edge, then e will reduce by three due to the two active corners changed to inactive and the insertion of the imaginary edge.

In case 4b, there will be no change in V because the normal gap merely transforms into a imaginary gap. H will increase by one which can be shown as follows:

$$
\begin{aligned}
H_j' &= \sum(n(Y_{j_k}) - 1 - g(Y_{j_k})) \\
&\quad -(n(Y_{j_a}) - 1 - g(Y_{j_a})) - (n(Y_{j_b}) - 1 - g(Y_{j_b})) \\
&\quad +n(Y_{j_a}) + n(Y_{j_b}) - 1 - g(Y_{j_{ab}}) \\
&= H_j + 1 - (g(Y_{j_{ab}}) - g(Y_{j_a}) - g(Y_{j_b})) \\
&= H_j + 1 - 0 \\
&= H_j + 1
\end{aligned}
$$

Similarly as in case 4a, when a special partition edge is involved, there will be no change in e and the increase in H merely compensates the decrease in spe. On the other hand, if there is no special partition edge, then e will decrease by three.

Case 4c is similar to the case 4a that has special partition edge between the corners. The only difference is that the special partition edge in case 4a is now just an edge caused by the overlapping

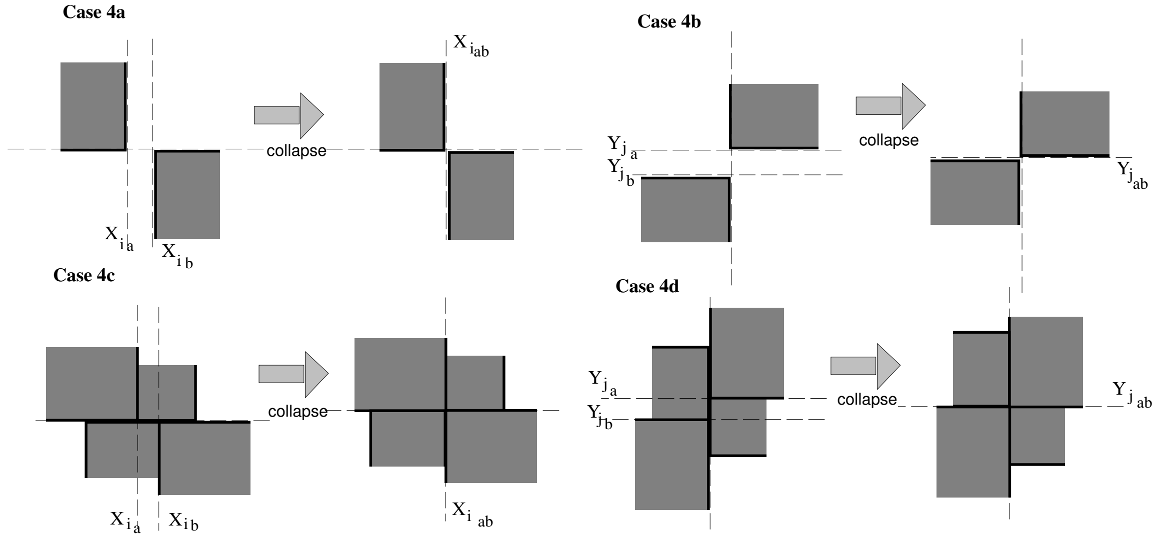


Figure 16: Collapse that introduces a imaginary gap

of line segments (sides of solid rectangles), i.e. there is no decrease in *spe* in this collapsing step. However, H does not increase either, and hence, there is no change in any parameters.

Again in case 4d, it is similar to case 4b with a special partition edge between the corners, except that the special partition edge is just an edge caused by the overlapping of line segments. However, V decreases by one due to the introduction of an imaginary gap. Hence, the increase in H compensates the decrease in V rather than *spe* as illustrated in case 4b.

We note that for case 4c and 4d, if anyone of the four rectangles is absent, or a pair of diagonal rectangles are absent (see Figure 17), the analysis of the changes to the parameters H, V, e and *spe* will not differ from their counterparts in Figure 16.

After the three stages, the perturbed partition would have changed back to the original partition. Hence, we have shown the invariance of Equation (5).

Finally, combining Equation (4) and Equation (5), we get

$$\begin{aligned} f &= \frac{(12n + 4 - 3V - 3H - 3spe) - 1}{3} - n \\ &= 3n + 1 - V - H - spe \end{aligned} \tag{6}$$

Hence, this completes the proof of Theorem 1. □

4 Optimum Partitioning

We now consider the problem of finding an optimum partition of a given layout. We first note that V and H are fixed for a given layout. Hence, by Theorem 1, to minimize the number of free blocks, we need to maximize *spe*, the number of special partition edges used. Also, the number of free blocks is not affected by how we select the remaining partition edges (namely, either vertical or horizontal).

Given any layout, let V_B and H_B denote the sets of all vertical and horizontal special partition edges, respectively. A vertical special partition edge $v_j \in V_B$ is said to *intersect* a horizontal special partition edge $h_k \in H_B$ if they share at least a point in common. However, by definition, the special partition edges in any partition of the free space do not intersect each other. Hence, the aim is to find a maximum set of non-intersecting special partition edges in $V_B \cup H_B$. For example in Figure 18 where all special partition edges are drawn, h_2 intersects v_1, v_2 and v_3. The maximum set of non-intersecting special partition edges is $\{h_1, h_2, h_3, h_4, h_5, v_5, v_6, v_7, v_8, v_9\}$. The optimum partition, which uses this maximum set of non-intersecting special partition edges, is shown in Figure 19.

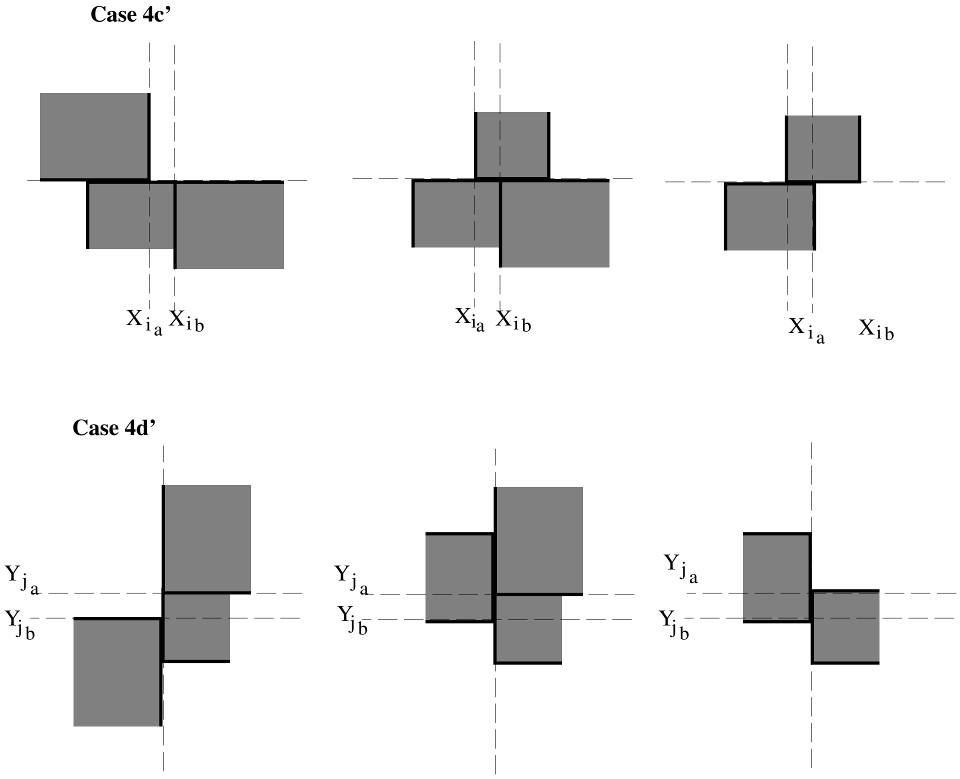
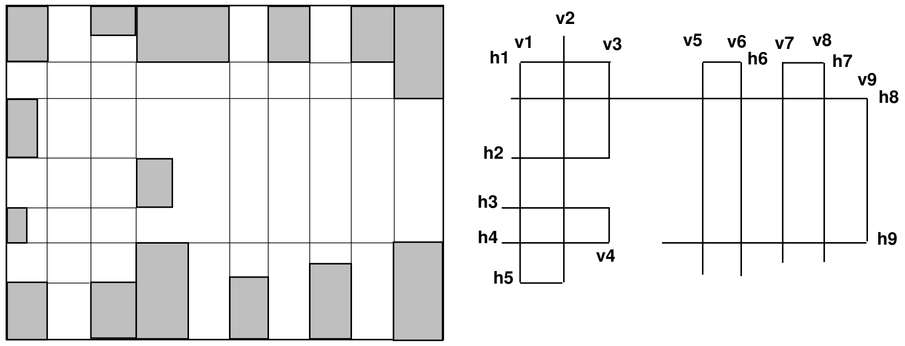


Figure 17: Collapse that introduces a imaginary gap, other cases

The problem of finding the maximum set of non-intersecting special partition edges can be transformed into a well-solved classical problem of finding maximum independent vertex set of a bipartite graph. We transform the set of all special partition edges into a bipartite graph $B(V_B, H_B, E_B)$ as follows: We let each vertical special partition edge v_i be a vertex $v_i \in V_B$ (for simplicity, we have used the same notation for the special partition edges and their corresponding vertices in the bipartite graph B) and each horizontal special partition edge h_j be a vertex $h_j \in H_B$, and there is an edge between vertices v_i and h_j if and only if the special partition edges v_i and h_j intersects. For example, Figure 20 is the bipartite graph for the set of special partition edges in Figure 18.

An independent vertex set of a bipartite graph $B(V_B, H_B, E_B)$ is a set of vertices such that no two elements in the set share common edges (i.e. independent set $I(B) \subseteq V_B \cup H_B$ and $n_i, n_j \in I(B) \Leftrightarrow (n_i, n_j) \notin E_B$). It is easy to see that an independent vertex set in the bipartite graph $B(V_B, H_B, E_B)$ corresponds to a set of non-intersecting special partition edges since the set of edges E_B in B represents intersections among the special partition edges. Therefore, the maximum independent vertex set of $B(V_B, H_B, E_B)$ is the solution to the optimum partitioning problem.



(a) (b)

Figure 18: Layout with all special partition edges

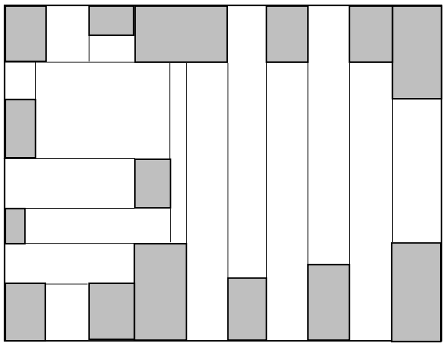


Figure 19: Optimum Partition

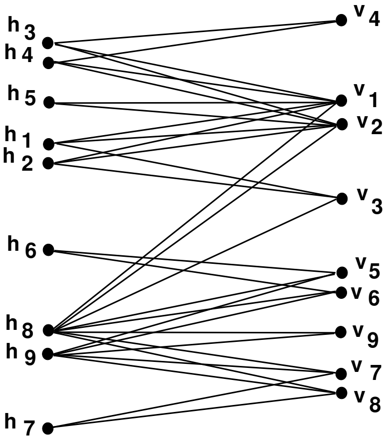


Figure 20: Bipartite Graph

5 Algorithm to find the Optimum Partition

The algorithm to obtain the optimum partition consists of three phases, namely, finding the set of all special partition edges $V_B \cup H_B$, finding the maximum set of non-intersecting special partition edges, and lastly building a data structure representing the partition.

In phase 1, we can obtain the set of all vertical and horizontal special partition edges by sweeping the plane horizontally and vertically respectively. A *priority search tree* [McC85] is used to keep track of active solid rectangles during each sweep for the purpose of checking if a gap cuts a solid rectangle. This phase can be done in $O((m+n)\ log\ n)$ where m is the number of all special partition edges.

Next in phase 2, using this set of special partition edges, we construct the bipartite graph $B(V_B, H_B, E_B)$. This can be done by a plane sweep on the set of special partition edges with the help of *priority search tree* in $O(m\ log\ m + K)$, where K is the number intersections. In order to find the maximum independent vertex set of B, we first need to obtain the maximum matching. This can be done in $O(\sqrt{m}K)$ [HK73]. Then, we could transform the maximum matching into a maximum independent vertex set as follows. Mark all unmatched vertices that belong to either V_B or H_B; use V_B without lost of generality. Mark all vertices that can be reached from a marked vertex by following an alternative sequence of an unmatched edge followed by a matched edge, starting with an unmatched edge, and this can be done by a modified *Breath-First search*. A maximum independent vertex set of B is the set of vertices in H_B that are not marked and the set of vertices in V_B that are marked. Since the operation is dominated by the computing of the maximum matching, phase 2 takes at most $O(m^{2.5})$.

Lastly in phase 3, two steps are involved namely, representing and placing of the chosen special partition edges obtain in phase 2 into the layout, and sweeping the resultant layout to build the remaining vertical partition edges and data structure which represents the partition.

In the first step, horizontal special partition edges could be placed at the respective locations as zero-height solid rectangles. Note that flat rectangles are considered as lines in the subsequent plane sweep. Vertical special partition edges need not be placed since we could let all other partition edges (including special partition edges) to be vertical in the next step.

In the second step, we adopt the $O(n\ loglog\ n)$ plane sweep algorithm proposed by [TAT91] which establish all vertical partition edges and build the data structure called *rectangulation graph*, where nodes are solid rectangles and free blocks, and edges represent left-right adjacency of the rectangles and free blocks. A source and sink can also be added (see Figure 21). However, the original sweep algorithm assumes that no sides of two rectangles has the same x or y value. We could not assume this as it is the key in minimizing the number of free blocks, and hence, the algorithm is extended to cater for such cases. In addition, the extended algorithm requires information about rectangles whose sides has the same y values (including flat rectangles). [1] For example, consider the rectangles in Figure 22. Let l_i and u_i denote the lower and upper horizontal sides of the solid rectangle R_i or S_i. Then the algorithm needs the relations: $left\_align(u_1) = u_2, right\_align(u_2) = u_1, left\_align(l_2) = l_3, right\_align(l_3) = l_2, left\_align(l_6) = l_4, right\_align(l_4) = l_6, right\_align(u_5) = u_6$, and $left\_align(u_6) = u_5$. Also, we are only interested in alignment of the same side of the rectangles, i.e. we do not have $left\_align(u_i) = l_j$ (or $right\_align(u_i) = l_j$) and vice versa. To obtain these relations, an additional pass to sweep the layout vertically is required. In this pass, we only need to sort the new zero-height rectangles and then merged with the original sorted list obtained in the vertical sweep of Phase 1. Each pair of neighbours is checked in constant time for alignment, and hence, this pass takes $O(n + m \log m)$.

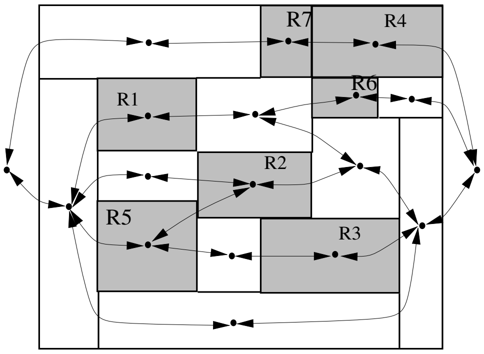


Figure 21: Rectangulation Graph with source and sink

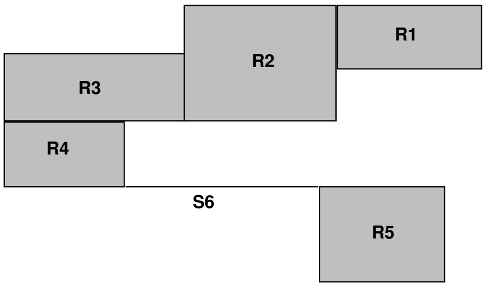


Figure 22: Illustration of alignment

In the final sweep, events arises when the vertical sweeping line enters (*insert* event) or leaves (*delete* event) a solid rectangle. In addition, we will resolve the conflict in the processing order if the events has the same x values as follows: process all the delete events in decreasing y values first before the insert events in increasing y values (y is upward increasing). For example in Figure 23, when the sweep line reaches the position shown, the order of processing is $delete(R_1)$, $delete(R_2)$, $delete(R_4)$, $insert(R_5)$, $insert(R_6)$, $insert(R_7)$ (Note that R_3 has no events at this sweep line).

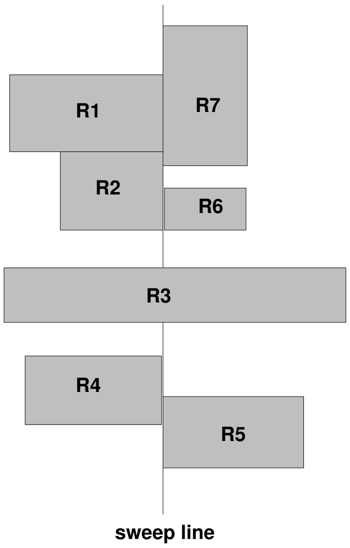


Figure 23: Illustration of processing order

The final sweep goes as follows. Upon entering a solid rectangle (insert event), the procedure $Insert()$ is called, and upon leaving a solid rectangle (delete event) the procedure $Delete()$ is called. These procedures are called with three parameters namely the upper, current and lower

[1] The extended algorithm does not need information about rectangles whose sides has same x values as these can be obtained readily during the sweep.

rectangle as they require to look-ahead and look-back in order to process the current rectangle. $Make\_Node()$ and $Complete\_Node()$ are procedures that are called within $Insert()$ and $Delete()$. The procedure $Make\_Node()$ is called to make a node in the rectangulation graph, where at the time of call, only the left (i.e. the sweep line) bottom and top boundary of the represented block is known. This new node is attached to the line segment (side of a solid rectangle) bounding the lower side of the block of this node. Edges that link this new node to those adjacent nodes on its left (that have already been completed) are also built. This is done by a keeping track of the completed nodes in the stack S. $Complete\_Node()$ simply updates the right boundary of the node that is created earlier and detached it from the line segment that it was attached to. It also puts the identifier of this new node into the stack S so that it can be linked to its right (while making those nodes on its right by $Make\_Node()$). The procedure $Pass\_Node()$ is used to pass a node attached to a line segment to another line segment. $Change\_Upper\_Node()$ and $Remove\_Node()$ are used to undo some mistakes made earlier due to insufficiency of information. They basically reverse the stack operations and dispose off edges built due to the wrong judgement of the upper boundary of the block. A *priority queue* [BKZ77] whose elements are upper and lower sides of the solid rectangles (line segments) is also used during the sweep. A line segment is inserted into the priority queue (i.e. made active) if it cuts the sweep line. Operations of the priority queue includes $Add()/Remove()$ which adds/removes an element, and $Pred()/Succ()$ which finds the active predecessor/successor of an element (i.e. the highest/lowest active line segment that is below/above the given line). The algorithms of $Insert()$, $Delete()$, $Make\_Node()$, $Complete\_Node()$, $Pass\_Node()$, $Change\_Upper\_Node()$, $Remove\_Node()$ and the main procedure $Sweep()$ are given in Appendix A.

Run time of this final sweep is analyzed as follows: Firstly, sorted list of the events can be obtained by first sorting the flat rectangles ($O(m \, log \, m)$), modify the event list from phase 1 ($O(n)$) and then merging the two list ($O(n+n)$). Aggregate time for all the calls on $Make\_Node()$ and $Complete\_Node()$ will take $O(n)$ as these two procedures only build the nodes and edges ($O(n)$ in number) with the help of a stack. Priority queue and stack operations takes $O(loglog \, n)$ and $O(1)$ each respectively. The run time of $Change\_Upper\_Node()$ and $Remove\_Node()$ depends on the number of nodes pushed back into the stack again. Since each node can at most be pushed back into the stack once, the cost is $O(n)$. Therefore, the final sweep takes $O(n \, loglog \, n + m \, log \, m)$.

The run times of the algorithms are summarized in table 1.

Phase 1	
Find all special partition edges	$O((m+n) \, log \, n)$
Phase 2	
Construct bipartite graph	$O(m \, log \, m + K)$
Find maximum independent set	$O(\sqrt{m}K)$
Phase 3	
Placing of special partition edges	$O(m)$
Finding alignment	$O(n + m \, log \, m)$
Plane Sweep	$O(n \, loglog \, n + m \, log \, m)$

Table 1: Summary of run times of algorithms - Optimum Partition

6 Discussion on a Similar Result

The same problem has been addressed in the paper [HLS93] in which a different approach is used to find the formula. As a result, a more complicated formula is given which depends on variables such as number of inactive corners, number of intersected (touching) solid rectangles, number of Manhattan edges and number of special partition edges. However, they have also established the fact that minimizing the partition requires maximizing the number of special partition edges.

Unfortunately, there is a flaw in their optimization algorithm. In section 6.1, we will discuss their algorithm that finds the maximum non-intersecting set of special partition edges. The discussion is based on the bipartite graph model rather than the line segments used in [HLS93]. Section 6.2 will present the flaw in the algorithm using some known results in graph theory. A counter example which exploits on the flaw is given in Section 6.3.

6.1 The "Algorithm"

The algorithm that is given in [HLS93] is as follows:

Algorithm to find Maximum Independent Vertex Set
(translated from [HLS93] to the bipartite graph model)

Step 1: Mark all vertices in V_B.

Step 2: Group vertices in H_B together if they have the same marked neighbors.
Lets call the groups $H_{B1\ldots k}$ such that
$H_{Bi} \cap H_{Bj} = \emptyset \Leftrightarrow i \neq j$
and $\bigcup_{i=1}^{k} H_{Bi} = H_B$.
Lets denote the set of marked neighbors of H_{Bi} as $\Gamma(H_{Bi})$.

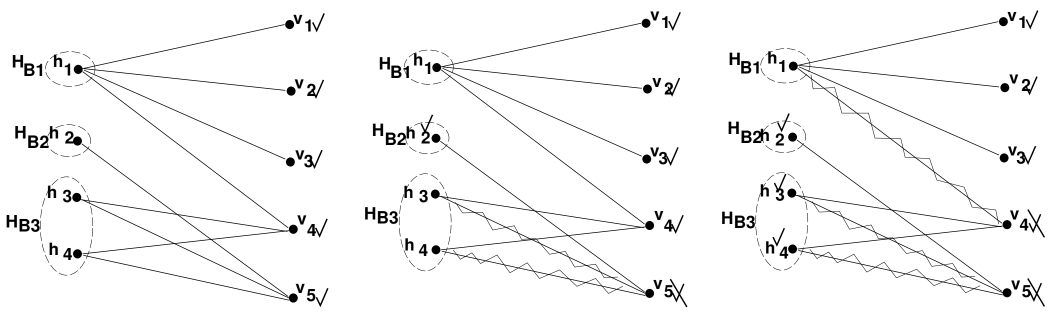
Step 3: Repeat until $|H_{Bi}| < |\Gamma(H_{Bi})|\ \forall i$
If $|H_{Bi}| \geq |\Gamma(H_{Bi})|$ then
Unmark all vertices in $\Gamma(H_{Bi})$.
Mark all vertices in H_{Bi}.

Step 4: Let V_{Bmax} denotes the set of marked vertices.

Step 5: Clear all marks and mark all vertices in H_B.

Step 6: Repeat step 2 and 3 by replacing V_Bs for H_Bs and vice versa,
and obtain the set of marked vertices denoted by H_{Bmax}.

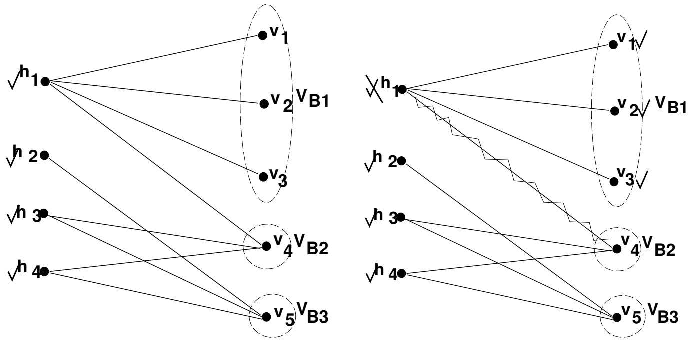
Step 7: Return larger set of V_{Bmax} and H_{Bmax}.



(a) (b) (c)

Figure 24: Example of obtaining V_{Bmax}

For example, Figure 24 and 25 demonstrate how the algorithm works. In Figure 24a, all vertices in V_B are marked and three groups H_{B1}, H_{B2} and H_{B3} are identified. Since $|H_{B2}| = |\Gamma(H_{B2})|$, v_5 is unmarked and h_2 marked (Figure 24b). With the unmarking of v_5, $\Gamma(H_{B3}) = \{v_4\}$, hence v_4 will have to be unmarked and h_3 and h_4 will have to be marked (Figure 24c). Now, we are left with H_{B1} and since $|H_{B1}| = 1$ and $|\Gamma(H_{B1})| = |\{v_1, v_2, v_3\}| = 3$, we stop the repeat loop with $V_{Bmax} = \{h_2, h_3, h_4, v_1, v_2, v_4\}$. We repeat the whole process by marking all vertices in H_B (Figure



(a) (b)

Figure 25: Example of obtaining H_{Bmax}

25), we should obtain $H_{Bmax} = V_{Bmax}$. Hence the algorithm returns the set $\{v_1, v_2, v_3, h_2, h_3, h_4\}$ as the maximum independent vertex set.

6.2 Flaw of the Algorithm

It is known that the complement of an independent vertex set is a vertex cover (i.e. the maximum independent set $MaxI(B) = (V_B \cup H_B) - MinC(B)$ where $MinC(B)$ is the minimum vertex cover of B). Hence, with reference to the algorithm, besides trying to obtain the maximum independent set $MaxI(B)$, which contains the marked vertices, it is trying to find the minimum vertex cover $MinC(B)$ as well, i.e. those vertices that are left unmarked. However, we can show that the algorithm may not give the minimum vertex cover as follows: Due to König [K$\ddot{3}$1], it is known that given any bipartite graph $B(V_B, H_B, E_B)$, the size of the minimum vertex cover $MinC(B)$ is equal to the size of maximum matching $MaxMatch(B)$, i.e. $|MinC(B)| = |MaxMatch(B)|$. Moreover, it is also shown in [K$\ddot{3}$1] that the $|MaxMatch(B)|$ is equal to $|V_B| - max(\delta(V_{Bsubset})$ $\forall V_{Bsubset} \subseteq V_B)$, where the deficiency function $\delta(V_{Bsubset}) = max(|V_{Bsubset}| - |\Gamma(V_{Bsubset})|, 0)$ and $\Gamma(V_{Bsubset})$ is the neighborhood set of $V_{Bsubset}$.

In the algorithm, due to the way V_{Bi}'s are constructed, $\Gamma(V_{Bi}) = \Gamma(V_{Bi_{subset}})$ $\forall V_{Bi_{subset}} \subset V_{Bi}$, and hence $\delta(V_{Bi}) > \delta(V_{Bi_{subset}})$ $\forall V_{Bi_{subset}} \subset V_{Bi}$. Hence, the algorithm is actually checking for deficiency for each group V_{Bi} and all its subsets $V_{Bi_{subset}}$'s. However, subsets of V_B that stretch over a few V_{Bi}'s are not considered. Since the algorithm flips the marks of vertices when a deficiency is detected, this fails badly if it happens that all groups do not have deficiency but a combination of such groups have positive deficiency. This applies for the sets H_{Bi}'s as well. As we will see in Section 6.3, the counter example that caused the algorithm to fail is based on this flaw.

6.3 A Counter Example

Exploiting on the flaw presented in the previous section, it is found that a bipartite graph that caused the algorithm to fail is realizable from a layout. Figure 19 is one such example. All special partition edges are drawn in Figure 18, and the corresponding bipartite graph $B(V_B, H_B, E_B)$ is shown in Figure 20. Figure 26 shows how the vertices are grouped together in the algorithm, and all the groups has no deficiency. Hence no marking or unmarking is done, and the algorithm returns either the set V_B or H_B as the "maximum" independent vertex set. However, we can see that the $MaxI(B) = \{h_1, h_2, h_3, h_4, h_5, v_5, v_6, v_7, v_8, v_9\}$, and hence the algorithm fails. It is obvious that the failure is due to H_{B1}, H_{B2} and H_{B3} not having deficiency but $H_{B1} \cup H_{B2} \cup H_{B3}$ has a deficiency of one. Similarly, for V_{B4}, V_{B5} and V_{B6} do not have deficiency, but $V_{B4} \cup V_{B5} \cup V_{B6}$ has a deficiency of one.

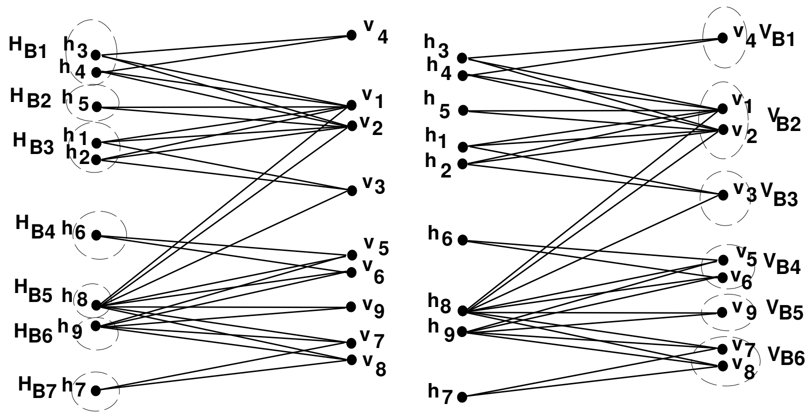


Figure 26: Bipartite Graph of Counter Example with the Groupings

7 Conclusion

Given a layout of rectangles in a rectangular region, we study the problem of partitioning the free space into disjoint rectangular free blocks. We present a formula to count the exact number of free blocks in a partition and an $O(n^{2.5})$ algorithm to obtain the optimum partition. In an independent work [HLS93], a more complicated formula was proposed together with an "algorithm" for obtaining the optimum partition. We have also shown that there is a flaw in their algorithm.

References

[BKZ77] P.V.E. Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical System Theory*, 10:99–127, 1977.

[BS87] J. Bhasker and S. Sahni. A linear time algorithm to check for the existence of a rectangular dual of a planar graph. *Networks*, 17:307–317, 1987.

[BS88] J. Bhasker and S. Sahni. A linear algorithm to find a rectangular dual of a planar triangulated graph. *Algorithmica*, 3:247–278, 1988.

[Gra68] J. Grason. A dual linear graph representation for space-filling location problems of the floor plan type. *Emerging Methods in Environmental Design and Planning (G.T. Moore ed), Proceedings of the Design Methods Group, First International Conference, Cambridge, MA*, pages 170–178, 1968.

[HK73] J.E. Hopcroft and R.M. Karp. A $n^{2.5}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Computing*, 2:225–231, 1973.

[HLS93] P.Y. Hsiao, C.Y. Lin, and P.W. Shew. Optimal tile partition for space region of integrated circuits geometry. *IEE Proceedings-E*, 140(3):145–153, 1993.

[K$\ddot{3}$1] D. König. Graph and matrices (in hungarian). *Mat. Fiz. Lapok*, 38:116–119, 1931.

[Kaj92] Y. Kajitani. A simple proof of the condition for floor planning. *JTC-CSCC'92*, pages B6–4–1 – B6–4–4, 1992.

[KK84] K. Kozminski and E. Kinnen. An algorithm for finding a rectangular dual of a planar graph for use in area planning for vlsi intergrated circuits. *Proc 21st DAC Albuguerque*, pages 655–656, June 1984.

[KK85] K. Kozminski and E. Kinnen. Rectangular duals of planar graphs. *Networks*, 15:145–157, 1985.

[Ku93] L.P. Ku. Algorithms for incremental bin-packing. *Honours Year Project Report, DISCS, NUS*, 1992/93.

[LK92] H.W. Leong and L.P. Ku. On the incremental 2-d bin-packing problem. *Proc. of Workshop on Two and Three Dimensional Spatial Data: Representation and Standards*, Dec 1992.

[LQ92] H.W. Leong and S.K. Quek. Geometric algorithms for rectilinear path problems and their applications. *SEARCC 92*, 1992.

[McC85] E.M. McCreight. Priority search trees. *SIAM J Computing*, 14(2):257–276, 1985.

[Ous81] J.K. Ousterhout. Caesar: An interactive editor for vlsi. *VLSI Design*, 2(4):34–38, 1981.

[Ous84] J.K. Ousterhout. Corner stitching: A data-structuring technique for vlsi layout tools. *IEEE Transaction on Computer-Aided Design*, CAD-3(1):87–100, 1984.

[Que90] S.K. Quek. Gridless routing for vlsi. *Honours Year Project Report, DISCS, NUS*, 1989/90.

[TAT91] T. Tokuyama, T. Asano, and S. Tsukiyama. A dynamic algorithm for placing rectangles without overlapping. *J. Information Processing*, 14(1):30–35, 1991.

A Algorithms

Sweep(L : list of sorted events (solid rectangles))

```
/*Initialization*/
Initialize stack S
Initialize Priority Queue
Add(boundary_bottom) /*Initialise priority*/
Add(boundary_top) /*queue with top and bottom of boundary*/
Make_Node(boundary_bottom, boundary_left, boundary_top)
Complete_Node(boundary_bottom, boundary_left) /* make a source node */
Make_Node(boundary_bottom, boundary_left, boundary_top) /*initialize the first node*/

current = Remove_List(L)
next = Remove_List(L)
prev = NULL

While curr ≠ NULL /*sweeping by processing curr*/
  If (next same x value and same side as current)
    ahead = next
  else
    ahead = NULL /*ie no look-ahead*/
  If (prev same x value and same side as current)
    back = prev
  else
    back = NULL /*ie no look-back*/

  If current is a left edge
    Insert(ahead, current, back)
  else
    Delete(back, current, ahead)
  prev = current
  current = next
  next = Remove_List(L)

If boundary_bottom just started
  Complete_Node(boundary_bottom, boundary_right) /*make a sink node*/
else
  Complete_Node(boundary_bottom, boundary_right) /*complete last node*/
  Make_Node(boundary_bottom, boundary_right, boundary_top)
  Complete_Node(boundary_bottom, boundary_right) /*make a sink node*/
```

```
Insert (R_a, R, R_b) /*R_a = Rectangle above; R_b = Rectangle below*/

l_a = lower edge of R_a
u_b = upper edge of R_b
l = lower edge of R
u = upper edge of R
currsweep = current sweep line

If (l not left aligned)
  If Pred(l) is made
    If Pred(l) not just started /*ie Pred(l) is not made at current sweep line*/
      If (R_b = NULL)
        If Pred(l) do not touch l /*check if R touch something below*/
          Complete_Node(Pred(l),currsweep) /*complete node on R's left*/
          Make_Node(Pred(l),currsweep,l) /*make a new node below R*/
        else
          Complete_Node(Pred(l),currsweep) /*complete node on R's left*/
      else
        If Pred(l) do not touch l /*check if R touch something below*/
          If Pred(l) ≠ u_b /*make sure R and R_b not touching*/
            Complete_Node(Pred(l),currsweep) /*complete node on R's left*/
            Make_Node(Pred(l),currsweep,l) /*make a new node below R*/
          else
            If (u_b is left aligned)
              Complete_Node(u_b,currsweep) /*complete node on R's left*/
              Make_Node(u_b,currsweep,l) /*make a new node below R*/
        else
          If Pred(l) ≠ u_b /*make sure R and R_b not touching*/
            Complete_Node(Pred(l),currsweep) /*complete node on R's left*/
          else
            If u_b left aligned
              Complete_Node(u_b,currsweep) /*complete node on R's left*/
    else /*ie a mistake may have occured*/
      If Pred(l) do not touch l
        If R_b = NULL or Pred(l) ≠ u_b or (Pred(l)=u_b and u_b left aligned)
          Change_Upper_Node(Pred(l),l) /*height of node need to be changed*/
      else
        Remove_Node(Pred(l)) /*node is built by mistake, dispose away*/

Add(u)
If ( u ≠ l ) /*make sure it is not zero-height rectangle*/
  Make_Node(l,currsweep,u) /*make node of solid rectangle R*/
  Add(l)

If (u not left aligned) and (Succ(u) and l_a) do not touch u
  Make_Node(u,currsweep, Lower(Succ(u),l_a)) /*make node above R*/
```

```
Delete (R_a, R, R_b) /*R_a = Rectangle above; R_b = Rectangle below*/

l_a = lower edge of R_a
u_b = upper edge of R_b
l = lower edge of R
u = upper edge of R
currsweep = current sweep line

If (u not right aligned)
  If (Succ(u) and l_a) do not touch u
    Complete_Node(u,currsweep) /*complete the node above R*/
else
  Pass_Node(u,right_align(u)) /*Node above R continues to its aligned neighbour*/

Remove(u)
If (l ≠ u) /*make sure it is not zero-height rectangle*/
  Complete_Node(l,currsweep) /*completes node of the solid rectangle R*/
  Remove(l)

If (l not right aligned)
  If (R_b = NULL)
    If (Pred(l) do not touch l) /*check if R touch something below*/
      Complete_Node(Pred(l),currsweep) /*completes node below R*/
      Make_Node(Pred(l),currsweep,Succ(u)) /*make a new node on R's right*/
    else
      Make_Node(Pred(l),currsweep,Succ(u)) /*make a new node on R's right*/
  else
    If (Pred(l) do not touch l) /*check if R touch something below*/
      If (Pred(l) ≠ u_b) /*make sure R and R_b not touching*/
        Complete_Node(Pred(l),currsweep) /*completes node below R*/
        Make_Node(Pred(l),currsweep,Succ(u)) /*make a new node on R's right*/
      else
        If (u_b) is right aligned) /*processing delayed if no alignment*/
          Complete_Node(Pred(l),currsweep) /*completes node below R*/
          Make_Node(Pred(l),currsweep,Succ(u)) /*make a new node on R's right; this node will be passed to right_align(Pred(l)) later*/
    else
      If (Pred(l) ≠ u_b)
        Make_Node(Pred(l),currsweep,Succ(u)) /*make a new node on R's right*/
      else
        If (u_b is right aligned)
          Make_Node(Pred(l),currsweep,Succ(u)) /*make a new node on R's right; this node will be passed to right_align(Pred(l)) later*/
```

Make_node($lower, currsweep, upper$**)**
Create a node $currnode$ with $lower, currsweep, upper$
as the bottom, left and top boundary respectively
Node_Attached($lower$) = $currnode$
(y_1, y_2) = vertical interval between $upper$ and $lower$
While $(y_1, y_2) \cap (y'_1, y'_2) \neq \emptyset$ where (y'_1, y'_2) = vertical interval of Top(S)
Link(Top(S),$currnode$)
If $y_2 \geq y'_2$
Pop(S)
Else
break from while loop

Complete_Node($lower, currsweep$**)**
$currnode$ = Node_Attached($lower$)
Update right boundary of $currnode$ with $currsweep$
Push($S, currnode$)

Pass_Node($lower_{old}, lower_{new}$**)**
$currnode$ = Node_Attached($lower_{old}$)
Node_Attached($lower_{new}$) = $currnode$)
Node_Attached($lower_{old}$) = nil

Change_Upper_Node($lower, upper$**)**
$currnode$ = Node_Attached($lower$)
Update the top boundary of $currnode$ with $upper$
(y_1, y_2) = vertical interval between $upper$ and $lower$
For each $backnode$ of $currnode$ starting from topmost
(y'_1, y'_2) = vertical interval of $backnode$
If $(y_1, y_2) \cap (y'_1, y'_2) = \emptyset$
Push($S, backnode$)
Unlink($backnode, currnode$)
else
exit (For loop)
If $y_2 \neq y'_2$
Push($S, backnode$)

Remove_Node($lower$**)**
$currnode$ = Node_Attached($lower$)
For each $backnode$ of $currnode$ starting from topmost
Push($S, backnode$)
Unlink($backnode, currnode$)
Node_Attached($lower$) = nil
Dispose($currnode$)